Checking Secrecy by means of Partial Order Reduction

C.J.F. Cremers and S. Mauw

Eindhoven University of Technology,
Department of Mathematics and Computer Science,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.

Abstract. We propose a partial order reduction for model checking security protocols for the secrecy property. Based on this reduction we develop an automatic tool that can check security protocols for secrecy, given a finite execution scenario. We compare this tool to several other tools.

1 Introduction

The transformation of our society into an information society proceeds faster than many have ever expected. Current society already relies heavily on networked information systems with the associated security risks. Digital information is not any more processed within a physically shielded environment, since networked computers are susceptible to attacks and information has to be conveyed over possibly insecure communication channels.

Such communications may represent a value, which can be of a direct or indirect nature. Purchasing e.g. a piece of music over the internet has a clear direct value, while the indirect value of establishing one's identity in an e-banking application may be even larger. This value of information is closely connected to the classical quality factors of information, which are *confidentiality*, *integrity* and *availability*. Viewed in a different way, these three factors are the possible *security goals* to be achieved by a security-aware application. The most studied security goals are *authentication* (which counts as a form of integrity) and *confidentiality* (or secrecy). In contrast to authentication, the notion of secrecy does not leave much room for different interpretations. However, there is still no definitive answer as to verify secrecy effectively.

Security protocols are communication protocols dedicated to achieving such security goals. They can play a role at application level (e.g. establishing a user's identity in an electronic auction), but also at lower network levels (such as IPSec, which is the internet protocol enhanced with security features for authentication and confidentiality). Experience shows that it is not hard to develop a security protocol that appears correct at first sight, but shows security breaches when assessed more thoroughly. The reason is that it is very hard to protect against all possible attacks of every possible intruder.

Several decades ago, it was recognized that formal analysis of security protocols is imperative to establish secure information systems. Although formal methods suffer from the well-known problem of scaling up to large systems, security protocols have the advantage of being rather small, at least in their abstract form. Many formal methods have been developed for or made applicable to the area of security protocols.

Although the relatively small size of a security protocol makes it amenable to formal analysis, the complexity of the verification problem makes computer support essential. Theorem provers and model checkers are the two major branches of tool support in this area. Theorem provers assist the user in constructing a formal correctness proof for the protocol. They often need human guidance during the verification process. Model checkers, on the other hand, process the provided input automatically. While theorem provers provide full evidence of the correctness of a protocol, model checkers often only increase the confidence in the protocol. A model checker searches through all possible behaviours (or rather, states) of the protocol and checks whether they all satisfy the required security property. In general this state space grows exponentially with the size of the input problem, or may even be infinite. Therefore, much research is performed on reducing the explored state space. A prominent approach is the so-called partial order reduction [1]. It makes use of

the fact that when exchanging two events in a trace it sometimes will not influence the property checked for in this trace. These traces are equivalent with respect to the checked property and only one of these equivalent behaviours has to be explored by the model checker.

The goal of this paper is to study the development of a dedicated model checker based on partial order reduction for verifying security protocols with respect to confidentiality. This may seem a rather trivial research goal, and indeed there are many general purpose model checkers that have been successfully applied to verifying secrecy in security protocols. However, we conjecture that these general purpose model checkers cannot make full use of the structure underlying the problem and thus are not fully optimized for this specific task. Therefore, the underlying assumption which we want to validate here is that a model checker dedicated to verifying confidentiality of security protocols will outperform a general purpose model checker instantiated for the chosen setting. We expect that studying the specific security goal, the specific intruder model, the specific agent execution model, the specific agent communication model, etc. gives insight in how to considerably reduce the explored state space. As a side effect, developing a dedicated model checker also allows us to develop a dedicated input and output format. This makes it easier to support formats that are most suited for displaying security protocols (and attack traces), such as Message Sequence Charts (MSCs). In this paper we make use of MSCs to express security protocols, scenarios and attack traces. The first two are manually crafted, but the attack traces are generated in an MSC format by our model checker.

This paper is structured in the following way. First we will discuss the general setting of security protocols in Section 2. In Section 2 we will show an algorithm for model checking secrecy, and propose a new algorithm. In Section 4 we discuss Scyther, a tool we developed on the basis of the new algorithm. Section 5 is reserved for discussing related work and some experiments. Finally, in Section 6 we indicate some future work and conclude.

Acknowledgments We thank Erik de Vink for the discussions on security protocol semantics and Ingmar Schnitzler for programming a prototype implementation of our model checking algorithm. Lutger Kunst is acknowledged for his contribution on displaying attack traces as Message Sequence Charts.

2 Security protocols

In this section we will review some security protocol terminology and present an example of a security protocol, the Bilateral Key Exchange Protocol (BKE, see [2]). The goal of BKE is that two parties agree upon a freshly generated secret key. Secrecy of this key is only one of the requirements of this protocol, but we will not discuss the other requirements.

Figure 1 shows this protocol in a Message Sequence Chart. The two vertical axes represent the two *roles* of the protocol, which are the initiator role I and the responder role R. We list the initial knowledge of each of the roles above the headers of the roles. Thus, the initiator has an asymmetric key pair (SKi, PKi) and knows the public key of the responder PKr. Likewise, the responder has asymmetric key pair (SKr, PKr) and knows the public key of the initiator. The way in which this initial knowledge was established is not made explicit. The initiator starts by creating a fresh nonce ni. This is a random, unpredictable value which is used to make the exchanged messages unique and thus helps to counter play-back attacks. The first message sent by the initiator consists of the pair ni, I which is encrypted with the public key of the intended responder, denoted by $\{ni, I\}_{PKr}$. Encryption is used to guarantee that only the intended recipient can unpack the message. Upon receipt of the message, the responder creates his own fresh nonce nr and a fresh symmetric key kir that he wants to share with the initiator. Goal of the protocol is to transfer this key to the initiator in a secret way. Therefore, the responder replies with the message $\{h(ni), nr, kir\}_{PKi}$. With this message he proves that he was able to unpack the previous message (by showing that he knows nonce ni, witnessed by sending a hash $h(ni)$ of this nonce). Furthermore, this message contains the key kir and a challenge nr. The complete message is encrypted with the public key of the initiator to ensure that only I can unpack the message. Finally, the initiator responds to R's

message by sending a hash of nonce nr encrypted with key kir. Herewith he acknowledges receipt of the previous message. At the end of the two roles we have listed the security claims as a special kind of event. Both participants claim that whenever they reach the end of their protocol the value of kir is not known to the intruder. The meaning of such an event is that in every trace of the system in which this event occurs the value of kir is not in the knowledge set of the intruder.

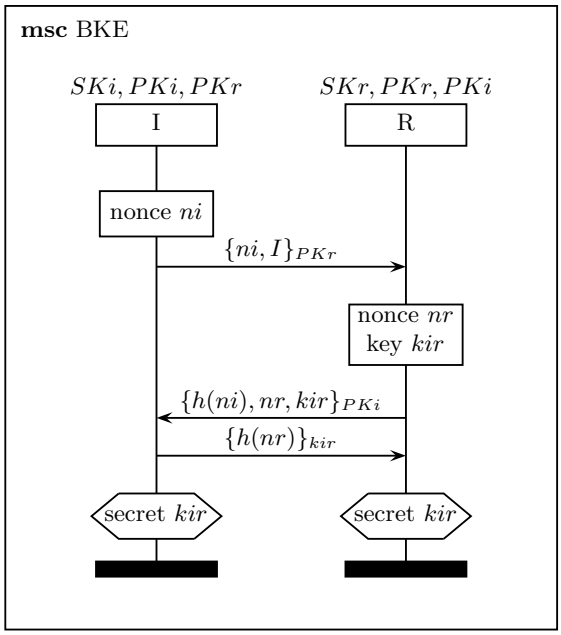

Fig. 1. The Bilateral Key Exchange protocol with public keys

A system executing this protocol consists of a number of agents, each of which may execute one or more instances of both roles (in parallel). When an agent executes a role from a protocol, we call this a run. Therefore, a system consists of a collection of runs exchanging messages to each other.

This simple model is not complete without describing the threats to which the system is exposed. Thereto, we assume the so-called Dolev-Yao intruder model (see [3]), which is considered the most general model of an adversary. This model implies that the intruder has complete control over the network and that he can derive new messages from his initial knowledge and messages received from honest agents. Hereby we assume that the intruder can only decrypt messages if he is in possession of the appropriate cryptographic key. Furthermore, we assume that a number of agents may conspire with the intruder and try to mislead the honest agents as to learn their secrets. Due to the capabilities of the intruder to intercept any sent message and to insert any message which can be constructed from his knowledge, we can model the existence of conspiring agents by assuming that their secret keys are in the initial knowledge of the intruder.

Now we come back to the Bilateral Key Exchange protocol. The specification from Figure 1 is correct if for any number of agents, executing any number of runs, in presence of a Dolev-Yao intruder, whenever an honest run enters a secrecy claim, the corresponding key kir is never exposed to the intruder.

We explain this in more detail using Figure 2. This Figure describes a sample scenario of the BKE protocol consisting of three runs, involving three agents a, b, and e, of which we assume that e conspires with the intruder. Thus we assume that the intruder has knowledge of the secret key SKe of agent e.

Each run is an instantiation of one of the two roles specified in Figure 1. The first run describes the behaviour of agent a performing the initiator role, expecting to be engaged in an execution of the BKE protocol with responder b. Notice that the incoming information (nonce nb and key kab) are stored in local variables (named U and K, respectively). The secrecy claim is therefore expressed with respect to the contents of variable K. The second run is also a run of agent a, performing the initiator role, but this time involved in a session with (conspiring) agent e. The third run is a run of agent b as a responder, involved with initiating agent a. In the second run, we did not include any secrecy claim. The reason for this is that if an agent starts a protocol session with an untrusted partner, his secrecy claim is bound to be violated and we will not consider this a protocol flaw. It is not required that within a scenario there is a matching responder role for every initiator role (or the other way around). The reason is that the intruder may abuse such non-matching runs to break secrecy of one of the involved keys.

Looking at the scenario, we see that there are three different types of events (ignoring the declaration of constants and variables). The first type is a send event. A send event may contain variables, but we require from the protocol that whenever a send is executed all its variables have already been assigned a value to. This does not hold for the second type of event, a read. Upon execution of a read event, its variables become bound to a concrete value. Finally, we have the secrecy claims, which, like a send event, have no uninstantiated variables upon execution.

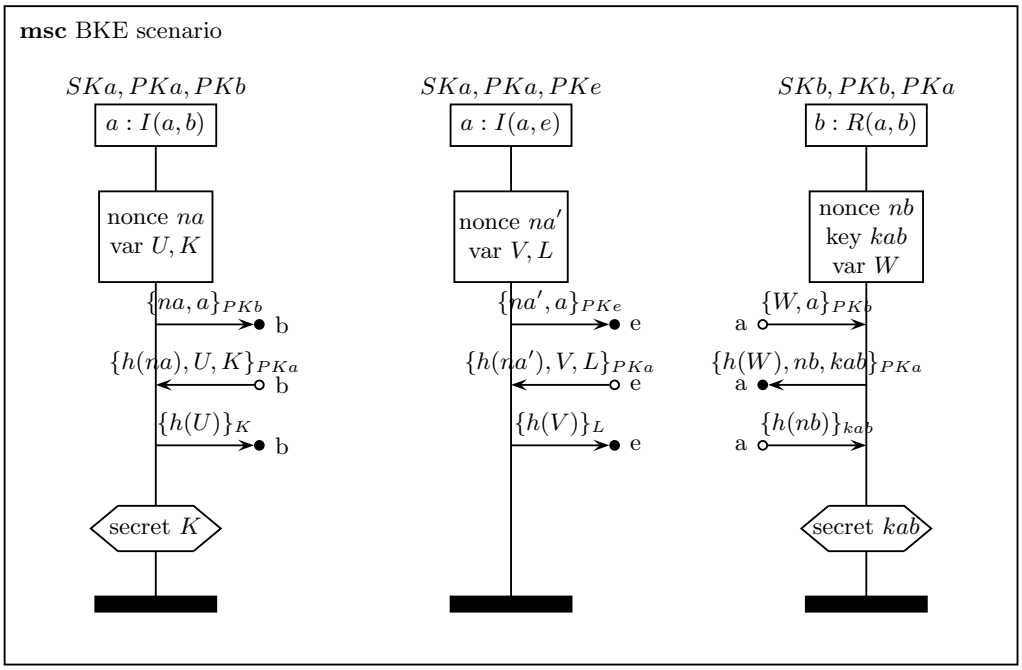

Fig. 2. Sample scenario for the BKE protocol

The events on each run are executed from top to bottom. Because of the chosen intruder model sending a message has no direct effect on a message reception. Execution of a send merely means that the contents of the sent message are added to the intruder knowledge. For instance, if the first run executes its first event, this means that the intruder learns message $\{na, a\}_{PKb}$. Since the intruder does not possess the corresponding key SKb he cannot unpack this message, so he can only store the complete message in his knowledge base. The intruder can decide to route this message through to the third run. Executing the first event of the third run then results in assigning value na to local variable W and the protocol proceeds as expected. If the second run sends its first message, the intruder will be able to unpack it and learn nonce na', which he may

be able to use to his advantage. Nevertheless, the secrecy claims of the first and third run will be valid in all possible execution orders.

In order to formally prove BKE correct for this scenario, we will have to check every possible interleaving of the included runs. And in order to prove BKE correct in general we have to do this for every finite scenario. Since this is rather complex we will explain how model checking can be used to prove correctness of BKE for a fixed set of runs.

3 Model checking secrecy in security protocols

We will develop our model checking algorithm in three steps. The first step yields a simple algorithm which naively searches through the complete state space. In the second step we transform this algorithm into an equivalent format that makes it possible to analyse which parts of the state space can be pruned. Finally, we present the reduced and efficient model checking algorithm.

In order to formulate our abstract algorithms we will need to formulate some of the notions from the informal explanation in the previous section a bit more precise.

intruder knowledge The knowledge of the intruder is modeled as a set of closed terms. If he learns a term, he can unpack it and learn all its sub terms. The only restriction is that he can unpack encrypted terms only if he knows the corresponding key. Adding a term to the intruder's knowledge is denoted by the operator $\oplus$.

enabled Executing a scenario boils down to executing the events of the runs in a given order. For each run we keep track of which event is to be executed next. This is the set of *enabled* events, with the restriction that a read event is only enabled if the intruder can construct a matching term. Whenever enabled, a send event and a secrecy claim contain closed terms, but a read event may still contain variables. When executing a read event its contents will be bound to a closed term.

match The *match* function determines which closed terms match a given read event. When developing the algorithms below we had no specific match function in mind as long as it yields a finite number of matching closed terms for every open term. It is a parameter of the algorithm. A straightforward match function could require corresponding types of all sub-terms, but a more lenient match could e.g. accept a nonce where a key would be expected and thus make the protocol susceptible to type flaw attacks. We will come back on type flaw attacks later.

after After executing an event, the system comes into a new state. Variables may become bound and the run whose event executes proceeds one step. The after function used below models this transition.

3.1 Algorithm

Algorithm 1 describes a simple depth-first search of all system states. Provided that the input scenario is finite and that the match function always returns a finite number of matching messages, this algorithm checks whether the protocol guarantees secrecy on the input scenario.

The recursive procedure traverseFull has three parameters: runs (the input scenario), know (the intruder knowledge which is a set of closed terms, initialised with the initial intruder knowledge), secrets (the set of closed terms claimed to be secret up to now, initialised with the empty set). This procedure works as follows. If there is a claimed secret known by the intruder, the algorithm halts, signaling the violation. Otherwise, for every run, it is checked whether its first event is enabled or not. All these enabled events can be executed in turn and we recursively check whether the sub-tree resulting after such an execution still guarantees secrecy. Since there are three types of events we have to determine the effect of each of these types on the state of the system. If the selected event is a secrecy claim, we know that its argument is a closed term and we add this term to the set of claimed secrets. If the selected event is a send event, we also know that its argument is a closed term and we add this term to the intruder knowledge. Finally, if the selected event is a read event, it might still have uninstantiated variables. Executing such an event means

that the intruder constructs a term from his knowledge, that matches the expected structure of the input term. Since we have to check correctness of the protocol for every possible behaviour of the intruder, we must recursively check every state resulting from every possible matching. This explains the for-loop in this case.

Algorithm 1: traverseFull (runs,know,secrets)

```
if any secret in know then
    exit ("attack")
else
    for all ev ∈ enabled(runs, know) do
        if ev = secret(m) then
            traverseFull(after(runs, ev), know, secrets ∪ {m} )
        end
        if ev = send(m) then
            traverseFull(after(runs, ev), know ⊕ m, secrets )
        end
        if ev = read(m) then
            for all m' ∈ match(know, m) do
                traverseFull(after(runs, read(m')), know, secrets )
            end
        end
    end
end
```

3.2 Transforming the algorithm

In the next step, we transform Algorithm 1 into an equivalent Algorithm 2, which generates exactly the same traces. The transformation mainly consists of replacing the outer for-loop by tail recursion. For this purpose, we use a choose function, that picks any element from the set of enabled events. An element is chosen, handled as in Algorithm 1, and the remaining elements from the enabled set will be handled by the recursive call. To make this possible, a new parameter $except : \mathcal{P}(Event)$ is added to the function. This set represents the events that were already selected by the choose function at this point in the trace construction. In this way, we have split the subtree in two parts, one containing the traces starting with the chosen event, and one containing the traces where this event is executed later. To facilitate this, we define a restricted enabled function that captures the remaining set of traces.

$$\mathsf{enabled2}(\mathsf{runs}, \mathsf{know}, \mathsf{except}) = \mathtt{enabled}(\mathsf{runs}, \mathsf{know}) \setminus \mathsf{except}$$

3.3 The refined algorithm

In the final step we reduce the number of traversed traces while retaining correctness of the algorithm. This results in Algorithm 3. The rationale for this reduction is that in many cases we can safely execute an event directly whenever it is enabled, while ignoring traces where this event occurs later.

The following lemma implies that whenever two closed events can be executed, it does not matter in which order they are executed. For send events this is trivial. For read events we use the fact that the intruder knowledge is non-decreasing, and for secrecy events we use the fact that the set of secrets is non-decreasing.

Algorithm 2: traverseFull2 (runs,know,secrets,except)

```
    if any secret in know then
    |  exit ("attack")
    else
    |  if enabled2(runs, know, except) ≠ ∅ then
    |  |  ev = choose(enabled2(runs, know, except))
    |  |  if ev = secret(m) then
    |  |  |  traverseFull2(after(runs, ev), know, secrets ∪ {m}, ∅)
*   |  |  |  traverseFull2(runs, know, secrets, except ∪ {ev})
    |  |  end
    |  |  if ev = send(m) then
    |  |  |  traverseFull2(after(runs, ev), know ⊕ m, secrets, ∅)
*   |  |  |  traverseFull2(runs, know, secrets, except ∪ {ev})
    |  |  end
    |  |  if ev = read(m) then
    |  |  |  for all m' ∈ match(know, m) do
    |  |  |  |  traverseFull2(after(runs, read(m')), know, secrets, ∅)
    |  |  |  end
    |  |  |  traverseFull2(runs, know, secrets, except ∪ {ev})
    |  |  end
    |  end
    end
```

Lemma 1. *Suppose that in Algorithm 1 at a given state closed events e and f from different runs can be executed. Then, after executing event e, event f can still be executed. Likewise, after f, event e can still be executed. Moreover, the states reached after ef and fe are both equal.*

This does not imply that in a trace any two events may be exchanged. If the second event gets enabled due to execution of the first event (e.g. if the first event is a send and the second is the corresponding read) it is not possible to execute the second event first. In conclusion, any event may be shifted towards the beginning of the trace until the first moment when it was enabled. The above lemma states that this can be done without changing the final state. Since the intruder knowledge and the set of secrets is monotonously non-decreasing we can simply discard the traces where this same event occurs later.

This motivates why we can simplify Algorithm 2 by omitting the recursive calls at the lines marked with $*$. Whenever a secrecy claim (which is a closed event) is enabled we can execute it and leave out the case where it is executed at a later moment. Likewise for a (closed) send event. Treatment of a read event is a bit more complex. Although it is safe to directly execute every closed instance of a read event and discard later occurrences of this instantiation of the read, it is erroneous to discard of every later occurrence of the read event. The reason for this is that the set of matching terms depends upon the intruder knowledge and that this knowledge may increase during the course of execution. Since these future matches are not possible at the current moment, we must allow the read to execute at a later moment too. However, we will then only have to consider new matches. All matches that are already possible in the current state must be avoided. To this end we introduce an extra parameter $\mathsf{forbidden} : \mathit{Event} \to \mathit{Knowledge}$ which assigns the current intruder knowledge to the considered read event. We limit the possible future matching to the cases which are not forbidden by this parameter. This is expressed in the following narrowing of the `enabled` set:

$$\begin{aligned}&\mathtt{enabled3}(\mathsf{runs}, \mathsf{know}, \mathsf{forbidden}) = \\ &\quad \{ev \in \mathtt{enabled}(\mathsf{runs}, \mathsf{know}) \mid ev = read(m) \Rightarrow \exists_{m' \in \mathtt{match}(know, m)} m' \notin \mathsf{forbidden}(ev)\}\end{aligned}$$

In the algorithm, we use $\mathsf{forbidden}[ev \to know]$ to denote the function that maps ev to $know$, and all events $e \neq ev$ to $\mathsf{forbidden}(e)$.

Algorithm 3: traverse (runs,know,secrets,forbidden)

```
if any secret in know then
    exit ("attack")
else
    if enabled3(runs, know, forbidden) ≠ ∅ then
        ev = choose(enabled3(runs, know, forbidden))
        if ev = secret(m) then
            traverse(after(runs, ev), know, secrets ∪ {m}, forbidden)
        end
        if ev = send(m) then
            traverse(after(runs, ev), know ⊕ m, secrets, forbidden)
        end
        if ev = read(m) then
            for all m' ∈ match(know, m) ∧ m' ∉ forbidden(read(m)) do
                traverse(after(runs, read(m')), know, secrets, forbidden)
            end
            traverse(runs, know, secrets, forbidden[ev → know])
        end
    end
end
```

4 The Scyther tool

Based on the proposed algorithm, we have developed a tool, called Scyther. Given a protocol and scenario description, this tool will construct a model and check it for the secrecy property.

The scenario consists of a number of runs. For each run, we define the protocol and role, and which agents are involved. The protocol and scenario descriptions are parsed by the Scyther tool and a model of the protocol with the scenario is constructed. This model is then checked for any violations of the secrecy property. Scyther enables the user to select various parameters such as the partial order reduction to be used, or any pruning behaviour when an attack is found.

When model checking an unknown protocol, choices have to be made for the scenario. With a small scenario, we will find a short attack quickly, but we can never be sure. On the other hand, a large scenario can take a long time to model check, even if there is a short attack. In many cases one would wish to have a breadth-first search, looking for short attacks first. Scyther's algorithm traverses the state space depth-first, to avoid any excessive memory requirements. To support some sort of breadth-first scan, Scyther allows for pruning the state space in the number of runs, or the maximum length of the traces. It can also automatically perform incremental state space searches, ranging over the number of runs or the maximum length of the traces.

4.1 Attack output

If an attack is found in the model, the tool generates an attack trace which is output in either ASCII format, or optionally in LaTeX format. Using the MSC macro package (see [4]) this can be automatically translated into an attack diagram. An example of the output can be found in Figure 3.

The presented attack is an attack to a slightly modified version of the BKE protocol from Figure 1. We construct a flawed protocol by replacing the last message $\{h(nr)\}_{kir}$ by $\{h(nr), kir\}_{PKr}$. This may seem a futile modification, but it introduces the possibility of a so-called man-in-the-middle attack. This attack can be exploited using the scenario from Figure 2. There are only two runs needed for this attack (see Figure 3): a run of agent a executing the initiator role in a session with compromised agent e, and a run of agent b as a responder to initiator a. The runs are numbered ♯0 and ♯1. This numbering is visible in the naming of the local constants. So kir♯1 means the (unique) key generated in run ♯1. The attack goes as follows. First, a sends an initialisation

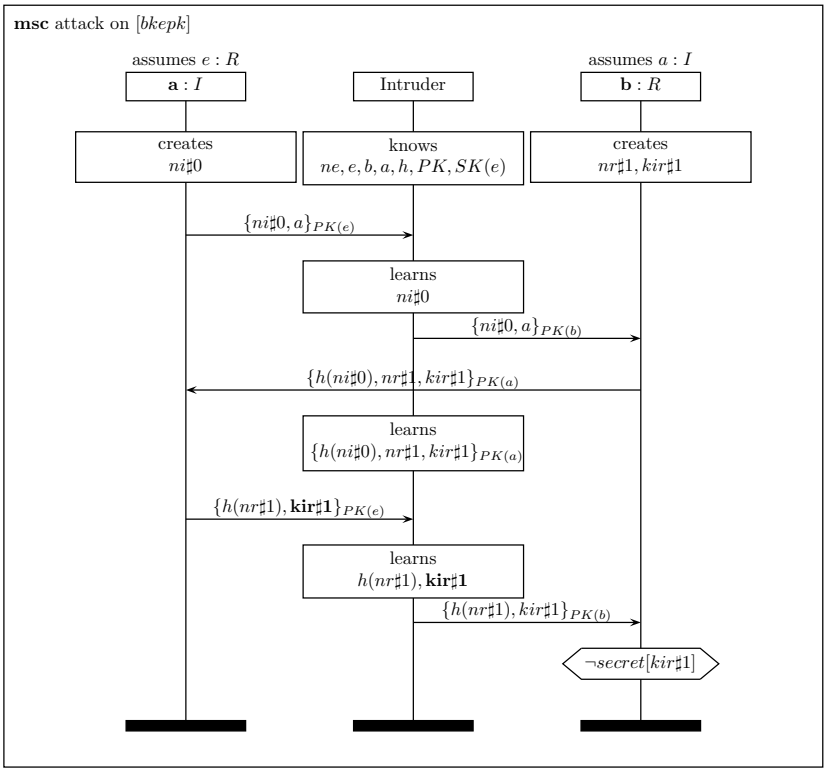

Fig. 3. Example output: an attack trace

message to e. This message is intercepted by the intruder and because the intruder knows SKe he can unpack the message and learn $ni\sharp 0$ (as expressed at the axis representing the intruder). The intruder then constructs a new message, encrypted with the public key of b and sends it through to b, forging that the message comes from a. Then b sends a message encrypted with PKa, which the intruder cannot unpack. However, he can use the run of agent a as an oracle to unpack this message, so he routes the message unmodified to a, forging sender e. After a's reply to this message, the intruder learns key $kir\sharp 1$, which is supposed to be secret for b.

4.2 Some internals

The model checking algorithm of Scyther mainly relies on term set manipulation. Send events add terms to the intruder knowledge, and read events try to match their patterns to the intruder knowledge. A large portion of the model checking time therefore relies on operations on the intruder knowledge. Any non-empty intruder knowledge is an infinite set, as it is closed under encryption and tupling. However, because it is constructed from the empty set only by adding terms, it can be represented by a finite set of terms.

Scyther is set up in such a way that it is easy to implement various partial order reductions and compare them. We have used this feature to generate the various test results.

5 Related Work and Experiments

There exist quite a number of security protocol model checkers. In this section we will discuss only a small selection of tools, and only those that operate on finite scenarios.

We have chosen to compare our work to the closely-related Brutus tool, the Casper/FDR toolchain, and a fairly recent development based on Constraint Logic programming. We will first

explain something about the differences between these tools and ours, before proceeding to some experimental results.

5.1 Brutus

A tool that is fairly comparable to ours is the Brutus tool [5]. As Scyther, it is based on partial order reductions, and also just explores a subset of all possible traces. However, instead of only checking for secrecy, Brutus can check for other properties as well.

Brutus reduces the set of traces on the basis of three observations:

- Internal events ordering.
- Send events ordering.
- Symmetry in the scenario.

The observations for the first two items can be explained as the fact that it is not relevant for the security property in which order some internal events occur. The same holds for send events: the order in which they occur can be neglected. In comparison to Brutus, Scyther uses a more powerful reduction. Scyther does not only consider the order of internal and send events, but that of all events.

For the third item, we have that in Brutus symmetry of the scenario is considered. Suppose in the scenario, there are two identical runs of the initiator role, with the same parameters. For checking secrecy it is not relevant which run executes its initial event first, as they are symmetrical. However, after the first event has occurred, the symmetry is broken. The gain of exploiting symmetry is considerably less than that of the other observations, as the authors of Brutus note in [5]. In fact, in the optimal case, the symmetry reduction used in Brutus decreases the number of states by a factor equal to the number of runs in the scenario. The current version of Scyther does not consider symmetry in the scenario.

Brutus uses a subset of ML as an input language for the protocol and the scenarios.

5.2 Casper/FDR

The toolchain Casper/FDR [6] is probably the most well-known tool in security protocol checking. It has been successfully applied to many protocols and has an extensive array of features. It can check for various security properties

Casper is a compiler that will translate a security protocol into a CSP process algebra model. The requirements are translated into another CSP model. The model checker FDR is then used to make sure that the protocol model is a refinement of the requirements model. Any optimizations that FDR applies to the model checking process are in fact general CSP refinement optimizations. It is therefore difficult to exploit the specific structure of the problem.

The main problem of using this toolset is that it cannot handle larger protocols, because large amounts of memory are required to execute the refinement check algorithm. The other methods that are discussed here exploit depth-first search, which is very efficient in terms of memory usage.

On the positive side, many protocols have already been checked using this toolset, and there is currently no other tool that can handle the same range of security protocols or range of possible requirements as Casper/FDR.

Casper uses a custom input language specifically tailored for security protocol checking. It is convenient for specifying a protocol, but we found that defining a very specific scenario can be cumbersome.

The output of FDR consists of an attack trace of the model that was generated by Casper. This can be interpreted again by Casper, to yield an attack trace on the same level as the one at which the protocol was specified. For some attack types, Casper can do a superficial analysis of the attack and give some hints on its nature, e.g. reporting that “Alice thinks she is talking to Bob” when authentication is violated. However, in many cases we found it difficult to interpret the actual attack.

5.3 Constraint Logic based approaches

The current developments in approaches based on Constraint Logic [7, 8] are based on the idea that the instantiation of variables in the construction of a trace might be postponed. In our model, variables are instantiated when they first occur in a trace. This results in a large number of branches in the model, one branch for each possible value of the variable.

We can try to postpone this instantiation as long as possible, by defining a constraint for each variable. We define a simple constraint as a predicate $v : K$, expressing that a variable v can be instantiated with any value from a knowledge set K. The Constraint Logic algorithm postpones instantiation of a variable when a read event is executed, and introduces a constraint for the message instead. After a complete trace is constructed, it is checked whether the constraints can be satisfied.

The algorithm on complete traces was suggested by Millen and Shmatikov in [7], where they also proved that it terminates. The constraint logic algorithm was later refined by Corin and Etalle [8] to allow for on-the-fly checking, which reduces the number of states traversed significantly. It also allows for checking of attacks in which partial runs are involved.

A strong point of this method is that it performs equally well for detecting a class of so-called type flaw attacks as it does for non-type flaw attacks. A type flaw attack occurs when a run is expecting to receive e.g. a nonce, but instead it gets sent an agent name, or a tuple (Key, Nonce). The constraint logic tools currently can detect the class of type errors with the restriction that tupling is taken to be non-associative. Scyther can only detect a small class of type flaw attacks, in which one basic type is mistaken for another basic type, e.g. when an agent name is mistaken for a nonce.

Both constraint logic tools use Prolog as input language for the protocol and the scenarios. The output of the tools is an uncommented attack trace. It gives almost no clues as to the nature of the attack, and it is up to the tester to try and reconstruct what has actually happened.

5.4 Experimental results

Given that the various approaches have different underlying models, it is quite difficult to objectively compare them. Minor choices in the protocol or the scenario result in performance changes that can be quite different for each approach.

We note that in general, experimental results are more often than not very poorly documented, and not reproducible. To make sure our experiments are reproducible we have set up a web page with all the data we used for our experiments at [9], to make sure that these results may be verified. We are aware that to give a complete analysis, it would be necessary to test the tools with several types and sizes of protocols, and invoke each of these with several scenarios. Here we will only test a single protocol, with scenarios that differ only in the number of runs defined.

The experiments serve two purposes. The first purpose is to determine the effect that our proposed algorithm has on the number of states that are traversed in the system. This directly affects the performance of the tool. The second purpose is to give some kind of performance comparison between our tool and others. We will address these two issues in the next sections.

Partial Order Reduction effectiveness One important validation of our algorithm is the effectiveness of the partial order reduction. As we have mentioned already, other tools such as Brutus also use some kind of partial order reduction. We claim however that our reduction is more effective.

In order to test this, we have conducted experiments with three different algorithms implemented within the Scyther framework.

As an inefficient algorithm, we implemented *traverseFull* as in Algorithm 1. We also implemented an algorithm that is as close as possible to the optimizations used by the Brutus tool, except that we did not implement the symmetry reductions. Finally, we implemented our improved traverse as in Algorithm 3.

Using these three algorithms we tested the Bilateral Key Exchange protocol for secrecy. As this protocol is correct, no attacks are found, and the complete state space is traversed. At the end of each test, the algorithms report the number of states traversed.

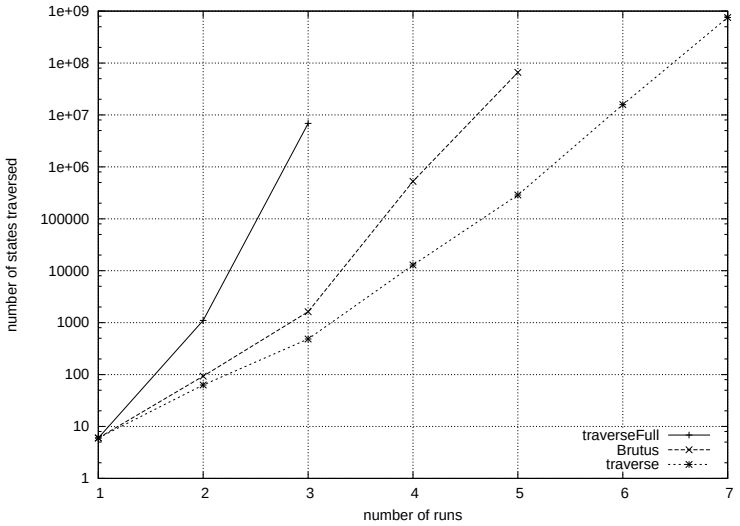

Fig. 4. State reduction

In Figure 4 the state space sizes are shown for each algorithm on the vertical axis, using a logarithmic scale. On the horizontal axis we have shown the number of runs in the scenario. When the running time of a test exceeded 24 hours, we stopped the test.

These experiments clearly show that the partial order reduction is effective for reducing the state space. Without any reductions, we could not test more than three runs in a day. The new algorithm allowed us to check a scenario with seven runs.

Performance Comparison The second type of experiment is meant to compare our tool with other tools, and is only measured in terms of computation time. Please refer to our web page [9] for the details of the machine used.

When model checking a scenario with a single run, or two runs, the algorithms are generally very fast. In some cases this caused the experiments to be immeasurable, reporting zero time usage. That is why we did not get sensible results for one run with Casper/FDR and with one or two runs with Scyther.

For Scyther and the Constraint Logic tool we aborted the test after running for 40 hours. In the case of Casper/FDR we did not reach this point, as the machine ran out of memory. This is inherent in the way the tools work. Scyther and the Constraint Logic tools use depth-first searches, and check each state for the secrecy property. This results in memory usage that is linear in the number of runs in the scenario, effectively making time the only limiting factor for checking large protocols. FDR on the other hand constructs two models, and tries to determine whether one model is a refinement of the other. This causes memory requirements for FDR to be exponential with respect to the number of runs.

The results are summarized in Figure 5. On the horizontal axis we again find the number of runs. The vertical axis is now the time taken by each test, on a logarithmic scale. We can see that Scyther outperforms the other two tools, and allows for checking larger models.

When we set up these experiments, we expected the constraint logic tools to outperform Casper/FDR. As it turns out, Casper/FDR gains significant performance at the cost of exponential memory usage.

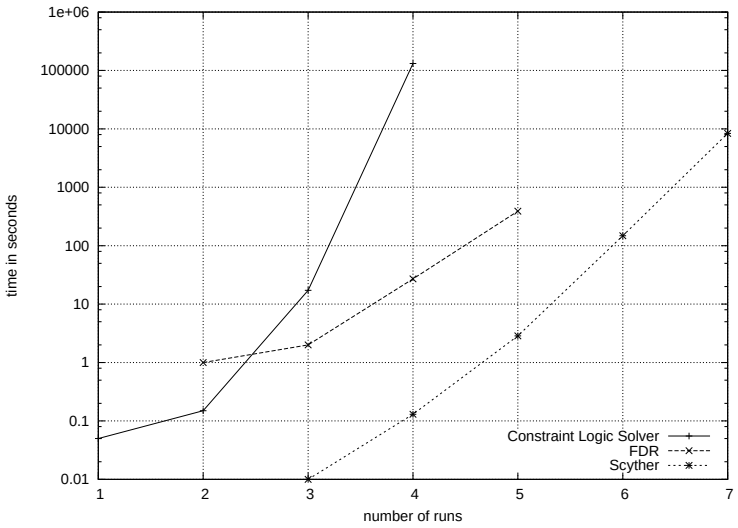

Fig. 5. Performance Comparison

6 Conclusions and Future Work

We have presented a new algorithm for model checking secrecy in security protocols, based on partial order reductions. The new algorithm significantly reduces the number of states that need to be traversed in model checking. As a results, it becomes feasible to check more intricate protocol scenarios as well as more extensive protocols.

The partial order reduction presented here was tailored specifically for the secrecy property in security protocol verification. For this property the reduction is sound and complete. However, there are many security properties, such as synchronisation [10] or agreement [11], for which this reduction is not complete. In the near future we will look into partial order reductions that are sound and complete for other security properties. We will investigate whether we can use these reductions to extend Scyther to be able to check a number of other security properties as well.

Furthermore, our proposed algorithm has a broader application than only for checking the secrecy property with a Dolev-Yao intruder. By varying over the match function, the contents of the system state and the security predicate, we are able to model check a wide range of security properties. We conjecture that correctness of our algorithm only requires the following two properties.

1. If closed events e and f can be executed, then after executing e event f can still be executed and the state reached after executing ef is equal to the state reached after executing fe.
2. The security predicate P is a monotonous predicate in the state of the system. By this we mean that if the system is in state σ and reaches state σ' after execution of any closed event, then $P(\sigma) \Rightarrow P(\sigma')$.

In this way we can e.g. strengthen the notion of secrecy as to require that trusted agents will also never learn the secret (unless they are explicitly allowed to). This generalization also makes it possible to vary over the intruder model. If we extend the state of the system with a buffer, containing sent messages and if we redefine the match predicate to take messages from this buffer rather than from the intruder knowledge, we have defined an intruder with just eavesdropping capabilities. Future work will examine the application scope of this algorithm.

In the scenarios checked, the constraint logic approach did not turn out to be very efficient in our tests. However, it has the advantage of being able to detect a class of type flaw attacks. We are currently investigating whether the constaint logic approach can be combined with the partial

order reduction. This might result in a more efficient algorithm for checking secrecy with type flaw attacks.

References

1. Peled, D.: Ten years of partial order reduction. In: Proceedings of the 10th International Conference on Computer Aided Verification, Springer-Verlag (1998) 17–28
2. Clark, J., Jacob, J.: A survey of authentication protocol literature. Technical Report 1.0 (1997)
3. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **IT-29** (1983) 198–208
4. Mauw, S., Bos, V.: Drawing Message Sequence Charts with LaTeX. TUGBoat **22** (2001) 87–92
5. Clarke, E., Jha, S., Marrero, W.: Partial order reductions for security protocol verification. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1785 of Lecture Notes in Computer Science., Springer (2000) 503–518
6. Lowe, G.: Casper: A compiler for the analysis of security protocols. In: Proc. 10th Computer Security Foundations Workshop, IEEE (1997) 18–30
7. Millen, J., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: ACM Conference on Computer and Communications Security. (2001) 166–175
8. Corin, R., Etalle, S.: An improved constraint-based system for the verification of security protocols. In Hermenegildo, M.V., Puebla, G., eds.: 9th Int. Static Analysis Symp. (SAS). Volume LNCS 2477., Madrid, Spain, Springer-Verlag, Berlin (2002) 326–341 `http://www.ub.utwente.nl/webdocs/ctit/1/00000096.pdf`.
9. Cremers, C.: Scyther documentation (2004) `http://www.win.tue.nl/~ccremers/scyther`.
10. Cremers, C., Mauw, S., de Vink, E.: Defining authentication in a trace model. In Dimitrakos, T., Martinelli, F., eds.: FAST 2003. Proceedings of the first international Workshop on Formal Aspects in Security and Trust, Pisa, IITT-CNR technical report (2003) 131–145
11. Lowe, G.: A hierarchy of authentication specifications. In: Proc. 10th Computer Security Foundations Workshop, IEEE (1997) 31–44